Application of Static Analyses for State Space Reduction to Microcontroller Binary Code

Bastian Schlich*, Jörg Brauer, Stefan Kowalewski

Embedded Software Laboratory, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany

Abstract

This article describes the application of two abstraction techniques, namely dead variable reduction and path reduction, to microcontroller binary code in order to tackle the state-explosion problem in model checking. These abstraction techniques are based on static analyses, which have to cope with the peculiarities of binary code such as hardware dependencies, interrupts, recursion, and globally accessible memory locations. An interprocedural static analysis framework is presented that handles these peculiarities. Based on this framework, extensions of dead variable reduction and path reduction are detailed. A case study using several microcontroller programs is presented in order to demonstrate the efficiency of the described abstraction techniques.

Key words: Static analysis, Model checking, Abstraction, Embedded software, Binary code

1. Introduction

Microcontrollers are often used in safety-critical systems for which correctness of software is crucial. Exhaustive testing of such systems in certain domains is not always performed due to fast time-to-market, uncertain environments, or the complexity of the systems. Model checking [10] is a formal verification technique that can be used to prove the correctness of software. In order to verify software for microcontrollers, we have developed a model checker for microcontroller binary code called [MC]SQUARE [1].

Model checking binary code has some advantages compared to model checking intermediate representations such as C code [4, 22, 25, 26, 36, 38]. Binary code is the code that is finally deployed to the microcontroller and not an intermediate representation. Therefore, all errors introduced during the complete development process such as compiler errors, reentrance errors, and errors in handling features of the microcontroller can possibly be found. Furthermore, the model checker does not have to account for the behavior of the compiler. All constructs in binary code have a clean and well-documented semantics and are easier to handle than some C constructs such as dynamic memory allocation and pointer arithmetic. Moreover, only the binary file of the program is needed and not the complete source code, which allows to check complete applications including external libraries.

Model checking of binary code, however, has two disadvantages. First, it is hardware dependent, and hence, has to be adapted for every microcontroller to be supported. Second, as more details are involved, the state-explosion problem [9] tends to be worse than when model checking C code. To apply model checking of binary code effectively, these two disadvantages have to be approached.

This paper describes how to tackle the state-explosion problem by applying two abstraction techniques, namely dead variable reduction (DVR) and path reduction (PR). While DVR merges states that only differ in values that are not read before they are overwritten, PR stores states only in program locations of interest. Both abstraction techniques

*Corresponding author

Email addresses: schlich@embedded.rwth-aachen.de (Bastian Schlich), brauer@embedded.rwth-aachen.de (Jörg Brauer), kowalewski@embedded.rwth-aachen.de (Stefan Kowalewski)

URL: http://www.embedded.rwth-aachen.de (Bastian Schlich)

[1] http://www.embedded.rwth-aachen.de/mc_square

are based on static analyses that annotate the program prior to model checking. These analyses have to cope with the peculiarities of microcontroller binary code such as hardware dependencies, interrupts, recursion, and globally accessible memory locations. This makes the application of intraprocedural approaches infeasible, and hence, an interprocedural approach that takes the underlying hardware into account is required. Both abstraction techniques were previously used in other model checkers (cp. Sect. 2), but due to the peculiarities of binary code, they could not be transferred to [MC]SQUARE one-to-one. This article, which is an extended version of a previously published paper [37], describes the application of these abstraction techniques for model checking programs written for the ATMEL ATmega16 microcontroller.

The remainder of this article is structured as follows. First, related work is presented in Sect. 2. Then, Sect. 3 details the challenges of applying static analysis to microcontroller binary code. In Sect. 4, a short introduction of [MC]SQUARE is given. Our interprocedural static analysis framework that is needed to cope with the peculiarities of binary code is detailed in Sect. 5. The described analyses form the basis of the two abstraction techniques DVR and PR, which are detailed in Sect. 6. The effectiveness of DVR and PR is demonstrated in a case study using several microcontroller programs in Sect. 7.

2. Related Work

Regehr et al. [32] describe an abstract interpretation for the analysis of stacks in microcontroller assembly code, for which a bit-wise representation is used. An analysis similar to our stack analysis described in Sect. 5.3.1 is described by Schwartz et al. [39]. Their approach does not check for physical addresses of the runtime stack but for certain properties, which are summarized as so-called *well-behavedness* in their approach. A particular challenge in the analysis of software for microcontrollers is the presence of interrupt handlers, which significantly differ from regular threads. A description of the main differences between threads in high-level programs and interrupt handlers is given by Regehr and Cooprider [31].

Yorav and Grumberg [45] describe DVR and PR for a parallel version of the *while* language, which is implemented for the model-checking tool MURPHI. In this language, every process has its own local variables, but global variables do not exist. Communication between processes is performed at fixed program locations by means of `send` and `receive` statements. For DVR, function calls are handled by inlining the body of the method at each call location. Hence, the static analysis can be performed intraprocedurally. So-called breaking points (cp. Sect. 6.2) used for PR can be determined completely statically since the language does not contain indirect control statements.

SPIN [17] uses both DVR and PR. It works on a language called *Promela*, which is similar to the parallel *while* language described before with respect to these two reduction techniques. This means that function calls are handled by inlining, communication is conducted at certain program locations, and indirect control is not present. Both analyses are performed statically via an intraprocedural approach prior to model checking.

Quiros [30] adapts the approach described by Yorav and Grumberg to a bytecode language used in a virtual machine. This bytecode language is similar to the parallel *while* language as it has no indirect control and communication between processes is performed at fixed program locations. The main important difference for static analysis is that the bytecode language used by Quiros has local and global variables, but they are easily distinguishable as different instructions are used to access global and local variables, respectively. DVR is only applied to local variables because the static analysis in this approach is performed intraprocedurally. The breaking points used in PR are determined statically as well.

Apart from SPIN and MURPHI, static DVR is integrated into numerous other model checkers such as BANDERA [11], IF [7], or BEBOP [5]. In particular, Bozga et al. [7] describe state-space reductions based on live variable analysis for the model checker IF. Furthermore, they show that abstractions preserving liveness establish an equivalence relation that is stronger than bisimulation. Their process algebraic specification of liveness is comparable to our definition of live variables for binary programs described later.

Another approach for DVR is used in the model checking tool ESTES as described by Lewis and Jones [23]. DVR is performed dynamically during state-space creation to exploit runtime information. Due to the dynamic nature of the approach, the results are more accurate in certain situations, but dynamic DVR increases the runtime. The user has to provide some information in order to use DVR such as a description of the behavior of the environment and addresses of the main function, interrupt handler starting points, and interrupt handler ending points. PR is not used

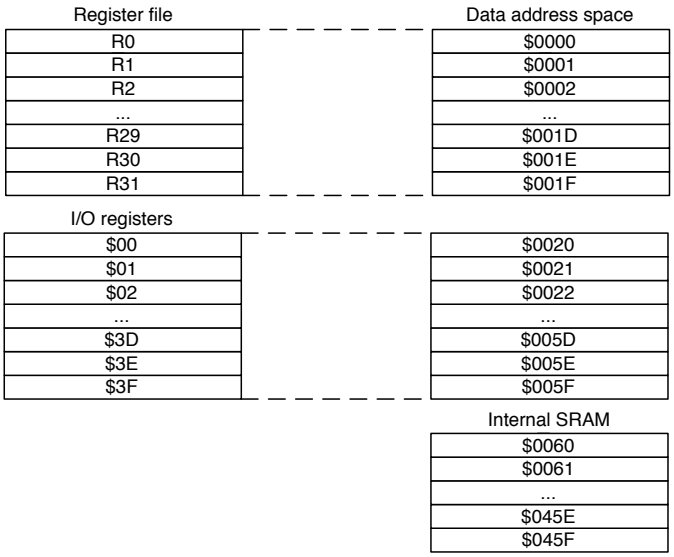


Figure 1: ATMEL ATmega16 data memory map [2]

in the ESTES model checker. An improved version of the dynamic algorithm of Lewis and Jones [23] is described by Self and Mercer [40]. Their approach eliminates the need for user interaction and produces maximal reductions for deterministic models, but is limited to single-procedure programs. This restriction makes their approach unsuitable for the analysis of binary code.

3. Applying Static Analysis to Microcontroller Binary Code

This section presents specifics of static analysis for microcontroller binary code. First, the microcontroller used in this paper is detailed, and then, challenges that arise when applying static analysis to microcontroller binary code are described. Solutions to these challenges are presented in Sect. 5 and Sect. 6.

3.1. ATMEL ATmega16 Microcontroller

The ATMEL ATmega16 microcontroller is an 8-bit microcontroller, which uses a Harvard architecture. In Harvard architectures, memory and buses for data and program are separated. A detailed description of the ATmega16 is given in its documentation [1, 2].

The ATMEL ATmega16 features 1120 bytes of data memory, 16 kB of in-system programmable flash memory, and 512 bytes of EEPROM memory. Figure 1 shows the organization of the data memory. The first 32 memory locations address the general-purpose registers, the I/O registers are addressed by the following 64 memory locations, and the internal data SRAM is addressed by the last 1024 memory locations. The general-purpose working registers are used, for example, in computations and to temporarily store values and the I/O registers are used to control peripherals of the microcontroller and to communicate with the environment. The ATmega16 features peripherals such as two 8-bit timers/counters, a 16-bit timer/counter, an analog-to-digital converter, and a watchdog timer. Communication is done by means of 32 I/O lines, which are organized in four 8-bit I/O ports. The internal data SRAM stores variables and the stack. The flash memory is used to store the program and the EEPROM is used to permanently store values.

The ATmega16 supports 131 instructions. As common for Harvard architectures, each of these instructions can only address one of the three memory types, that is, there are different instructions to access the data memory, the flash memory, and the EEPROM memory. The ATmega16 supports direct and indirect addressing modes. The indirect addressing mode depends on the instruction and uses either one of three 16-bit pointer registers (`X`, `Y`, and `Z`) or the stack pointer, which is located in two adjacent 8-bit registers `SPL` and `SPH`.

Interrupts are an important feature of microcontrollers. The ATmega16 supports 21 different interrupts. Operation of the interrupts is controlled by certain I/O registers. The global interrupt flag, which is located in the status register (SREG), controls whether interrupts are enabled. Each interrupt has an extra flag, which is located in an interrupt control register, that determines whether the specific interrupt is enabled. Additionally, many interrupts have an interrupt source, for example, timer interrupts depend on the corresponding timer/counter. That is, these interrupts can only occur if the corresponding source is active. Interrupts have fixed priorities, which are only important if two interrupts occur at the same time. In this case, the interrupt with higher priority is handled first. There are two modes for interrupt handlers, which can either be interruptible or non-interruptible. In the interruptible case, interrupt handlers are interrupted by all other interrupts including interrupts of lower priority. In the non-interruptible case, interrupt handlers are not interruptible at all. The standard behavior is that interrupt handlers are non-interruptible. Developers have to enable interrupts within interrupt handlers in order to make them interruptible.

For each interrupt there is an interrupt vector, which points to the corresponding interrupt handler. All interrupt vectors are combined in the interrupt vector table, which can be placed at different locations in memory depending on the configuration of two bit-fuses. By default, the interrupt vector table is located at the lowest addresses of the program memory. Interrupt vectors are ordered from higher priorities to lower priorities in the interrupt vector table.

3.2. Challenges

Applying static analysis to binary code involves some challenges because of binjary code constructs that make a generic intraprocedural static analysis approach infeasible. Binary code is hardware-dependent as each microcontroller architecture has its own instruction set and hardware features. Additionally, in microcontroller binary code there are instructions to access specific registers that change the behavior of the microcontroller or influence other registers. For example, interrupt control registers enable or disable interrupts, timer control registers enable or disable timers, and certain output registers change values of specific input registers. These hardware dependencies have to be accounted for in the static analysis.

All memory locations are globally accessible in binary code. This includes registers used for indirect accesses and indirect control. An important feature in binary code is the stack. It is used for different purposes such as exchanging values, saving the contents of the status register, and storing return addresses resulting from function calls or the execution of interrupt handlers. The stack is accessed using specific instructions `push` and `pop`, which utilize the stack pointer, but it can also be accessed directly.

Functions are not explicitly defined in binary code. All program locations can be the target of a `call`, `rcall`, or `icall` instruction. Hence, all instructions that are targets of a `call` instruction have to be handled as entries to functions. Often, different functions share some common code fragments. Functions are left using `ret` instructions. Interrupt handlers are implicitly defined in binary code. They are entered via `jmp` instructions from the interrupt vector table and left via `reti` instructions. The `reti` instruction differs from `ret` in that it enables interrupts upon execution while `ret` leaves the global interrupt flag unchanged. Interrupt handlers are similar to functions, but in contrast to functions, interrupts can occur at all program locations where they are enabled. Therefore, an analysis is required that determines program locations where interrupts are enabled.

Interrupts introduce pseudo-parallelism because they can interrupt the main program at all program locations where they are enabled, but the main program cannot interfere with interrupt handlers. Moreover, interrupt handlers exchange information with many program locations as all memory locations are globally accessible and interrupts can occur at many program locations. Interrupts, as well as recursion (which is not recommended but frequently found in microcontroller binary code), render the application of interprocedural approaches that use inlining or call-strings useless. Handling functions by assuming that they change all variables is also not appropriate as this over-approximation is too coarse to obtain meaningful results. These challenges require an interprocedural approach that has to cope with globally accessible memory locations, interrupts, and recursion. This approach has to incorporate hardware dependencies in order to handle the specifics of the respective microcontroller.

4. [MC]SQUARE

[MC]SQUARE stands for *model checking microcontrollers*. It is a discrete Computation Tree Logic (CTL) [14] model checker for the verification of microcontroller binary code. It supports binary code for several microcontrollers including the ATMEL ATmega16, the ATMEL ATmega128, the Infineon XC167, the Intel MCS-51, and the Renesas

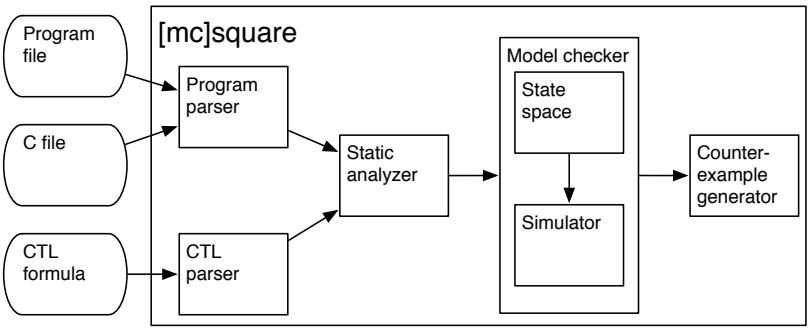


Figure 2: Model checking process in [MC]SQUARE

R8C/23. Moreover, [MC]SQUARE supports model checking software for Programmable Logic Controllers (PLCs) written in IL [35] and abstract state machines [6]. The CTL model checking algorithm used in [MC]SQUARE is a local model checking algorithm that was first proposed by Vergauwen and Lewi [43] and later adapted by Heljanko [16]. This algorithm can be applied on-the-fly during model checking. Thereby, [MC]SQUARE can locate errors in programs that are too large to be checked completely.

[MC]SQUARE takes as input a binary file, the corresponding C code if it is available, and a specification given in CTL. It supports different binary file formats such as Executable and Linking Format (ELF), Intel Hex, and Motorola S file format. For some of these formats, it processes debug information to relate binary code and C code. Within the CTL formula, users can make propositions about registers, I/O registers, memory locations, C variables, and the program counter. Additionally, [MC]SQUARE checks for stack collisions, stack overflows, and unintended use of microcontroller features such as write accesses to reserved registers.

The fundamental concept of the approach implemented in [MC]SQUARE is to use tailored simulators to build state spaces for model checking. These simulators utilize domain-dependent information during state-space building. This information is used to minimize state spaces, to allow propositions about all features of the system to be verified, and to present counterexamples in a representation that is understood by users. Within these simulators, our main interest is the development of both domain-dependent and domain-independent abstraction techniques in order to tackle the state-explosion problem.

[MC]SQUARE allows state spaces to be stored in main memory and on hard disk. State spaces can be built using single processors, multiple processors, or multiple computers. Counterexamples and witnesses are presented in the disassembled binary code, in the control flow graph of the disassembled binary code, in the C code, and as a state space graph. A detailed description of [MC]SQUARE and its theoretical foundations is given by Schlich [34].

Figure 2 shows the model checking process applied in [MC]SQUARE. The general process is the same for all supported microcontroller architectures. In this process, first the binary program file, the C file, and the CTL formula are read and transformed into their internal representations. Then, the static analyzer is executed and the program is annotated. The annotations are used by abstraction techniques implemented in the simulator to reduce the state space during model checking. For the ATMEL ATmega16, several static analyses such as live variables analysis, reaching definitions analysis, and analysis of interrupt registers are implemented. Details are presented in Sect. 5 and Sect. 6.

In the next step, model checking is started. The model checker obtains the initial state from the state space and evaluates the validity of certain subformulas of the current formula in this state. Then, depending on the result, it requests successors of this state from the state space and continues checking subformulas. As the model checker uses a local algorithm, it does not determine the truth values of all subformulas in all states. It only determines the truth values that are needed to determine the truth value of the overall formula in the initial state. If the model checker requires successors of a state that are not created yet, the state space uses the simulator to create successors on-the-fly. In order to create successor states, the simulator conducts the following four steps:

1. Load state into the microcontroller model
2. Determine all possible assignments for nondeterministic values
3. For each such assignment

(a) Simulate the effect of the next instruction
(b) Evaluate truth values of atomic propositions

4. Return resulting states

First, the state is loaded into the model of the microcontroller used within the simulator. Then, the simulator determines which nondeterministic values have to be instantiated. Nondeterministic values are introduced by the microcontroller model. In the model of the ATmega16, for example, accessing a timer, reading input from the environment, or handling of external interrupts introduces nondeterminism. Modelling timers with deterministic behavior leads to state explosion, and thus, [MC]SQUARE resorts to nondeterminism whenever timer-values are accessed.

In order to correctly handle nondeterminism and to guarantee the validity of the model checking results, the simulator has to create an over-approximation of the behavior shown by the real microcontroller. This is achieved by assigning all possible combinations to nondeterministic values accessed in the instruction to be executed next.

For example, executing an `add` instruction, which sums up two deterministic values, creates a single successor as no value assignment is needed. If in the same program location an external interrupt is enabled, two successors are created. In one successor the interrupt handler is entered, and in the other successor the `add` instruction is executed. Another example is the execution of an instruction that reads input from the environment by means of an I/O port. If an 8-bit port is used for input, 256 successors are created as all 256 possible values have to be assigned. For each of these assignments, the simulator executes the effect of the next instruction and evaluates the truth values of atomic propositions for the resulting states. This leads to a state explosion, but [MC]SQUARE uses several automatic abstraction techniques such as *delayed nondeterminism* [29] that tackle this problem within the simulator. Thus, the actual model checking algorithm does not have to account for hardware-specific information as this information is kept within the simulator. When all successors are created, they are added to the state space. After model checking is finished, the counterexample generator is invoked and presents the counterexample in the different representations available.

The counterexample generator takes the result and builds a counterexample or a witness depending on the formula and the outcome of the model checking process. It presents the counterexample in all existing representations, and users can choose the representation that fits their needs best. In all these representations, users can step forward and backward through the counterexample and analyze the values of registers, I/O registers, general memory locations, variables, and the program counter. This makes it easy for users to locate the source of errors found. The capability of stepping both forward and backward renders [MC]SQUARE a useful tool for debugging.

5. Static Analysis

This section explains the static analysis framework implemented in [MC]SQUARE. Due to the nature of binary code, some preliminary analyses are required to make DVR and PR applicable. These preliminary analyses are stack analysis (STA), reaching definitions analysis (RDA), global interrupt flag analysis (GIFA), and live variable analysis (LVA).

STA detects dependencies between values stored on the stack and values read from the stack by tracking corresponding pairs of `push` and `pop` instructions. Often, the value of a register is not changed in a called function because it is stored on the stack and restored later. RDA determines for each program location and each memory location where the memory location may have been assigned a value. Together with RDA, an extended constant propagation analysis is performed. GIFA applies an abstract interpretation to infer the value of the interrupt flag at each program location. LVA determines for each program location the memory locations that may be read on some path through the program before they are overwritten.

STA evaluates each function on its own, and hence, it is executed as an intraprocedural analysis. In contrast, the other three analyses are executed using interprocedural fixed point iterations. In the following, first the general approach for intra- and interprocedural fixed point algorithms used in [MC]SQUARE is described. Then, the hardware model implemented in [MC]SQUARE is presented before the static analyses are detailed in their order of execution.

5.1. Approach

In the context of [MC]SQUARE, both intra- and interprocedural analyses are required, which – depending on the property of interest – are executed as forward or backward data-flow analyses [28]. This section focuses on forward analyses, where information about the program execution is propagated along the edges of the control flow graph

(CFG) of a program. A backward analysis can be seen as the dual as it operates on the reversed CFG. Information about the program is represented using finite lattices [13]. In the following, we refer to elements of the respective lattices as *data-flow facts*. This section first gives an overview of our general approach. It then explains a standard intraprocedural fixed point algorithm before detailing our approach to interprocedural analysis in presence of recursion and global variables.

5.1.1. Overview

In [MC]SQUARE, the different static analyses interact as depicted in Fig. 3. First, STA is executed as it forms the basis of all other analyses. The major dependency is between RDA and GIFA. The constant propagation executed together with RDA influences the precision of GIFA and the status of the global interrupt flag is required in order to take the execution of interrupt handlers into account during RDA. In the end, LVA is executed as it depends on the results of GIFA.

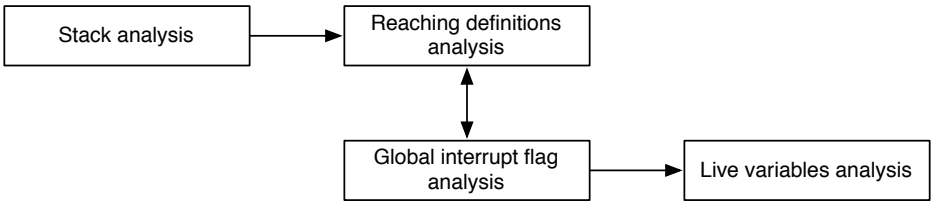


Figure 3: Dependencies between the different static analyses implemented in [MC]SQUARE

In order to handle the mutual dependencies between RDA and GIFA, these two analyses are executed in alternating order until the results of one of the analyses remain unchanged as depicted in Fig. 4. That is, a fixed point iteration of different analyses is conducted. In the first iteration, the execution of RDA is based on the assumption that interrupts are active in all program locations, generating a coarse over-approximation. These results are then used in GIFA in order to obtain a more precise over-approximation of the status of the global interrupt flag. In the second iteration, these more precise results of GIFA are used to obtain more precise RDA results, which are then used in GIFA, and so forth. If RDA is executed, the results of RDA from the previous iteration are stored to detect the fixed point, but they are not used for the computation, that is, all data-flow facts are reset in the beginning of each iteration. After each iteration, the new results are compared to the results of the previous iteration. If they are equal, a fixed point is reached and the iteration terminates. Otherwise, another iteration of GIFA is executed with more precise results from RDA. The same applies for GIFA, that is, GIFA uses results of previous iterations only for detecting fixed points. Thus, the analysis results become more precise with each iteration of RDA and GIFA and eventually remain unchanged.

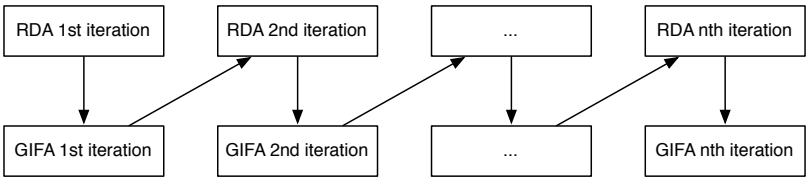


Figure 4: Execution of RDA and GIFA

The following describes this approach formally. Here, the results of RDA and GIFA after the i-th iteration are denoted by RDA^i and $GIFA^i$, respectively. Then $RDA^{i+1} \sqsubseteq RDA^i$ and $GIFA^{i+1} \sqsubseteq GIFA^i$. Since the results are monotone decreasing with each iteration and the domains are finite, there exists $n \in \mathbb{N}$ such that $RDA^n = RDA^{n+1}$ and $GIFA^n = GIFA^{n+1}$. That is, the analysis results eventually stabilize and the iteration terminates. From our experience, this is typically the case after the third or fourth iteration.

5.1.2. Intraprocedural Analysis

Given the CFG $G = (V, E)$ of a program and a lattice L representing data-flow facts, a data-flow analysis can be expressed in terms of an equation system over program locations $p \in V$. Here, a monotone transfer function

$\omega : V \times L \to L$ computes the effects of executing an instruction on the data-flow facts. That is, given a program location p and $l, l' \in L$, it is $l \sqsubseteq l' \Rightarrow \omega(p, l) \sqsubseteq \omega(p, l')$. The monotonicity of ω in combination with finite lattices ensures termination of the fixed point iteration. The following equation system expresses a forward data-flow analysis, that is, information is propagated along the edges in the CFG:

$$\begin{aligned} dfa_{entry}(p) &= \begin{cases} \bot & : p \text{ is initial} \\ \bigsqcup\{dfa_{exit}(p') | (p', p) \in E\} & : \text{otherwise} \end{cases} \\ dfa_{exit}(p) &= \omega(p, dfa_{entry}(p)) \end{aligned}$$

Such an equation system can, for instance, be solved using a worklist-based fixed point iteration. This is a well-known approach, which was extensively studied in the past [15, 18, 24, 28].

5.1.3. Interprocedural Analysis

As argued before, the analysis of binary code requires interprocedural analyses (cp. Sect. 3). While it is possible to encode call-edges in the equation system in order to perform interprocedural analysis, this approach leads to loss of context-sensitivity. That is, it is not possible to distinguish different behaviors for different call-sites, and the analysis results are unified, leading to loss of precision. Other techniques such as inlining and the call-string approach [41, 44] are unsuitable due to the presence of unbounded recursion.

In order to deal with recursive function calls and to implement a context-sensitive analysis, we developed an interprocedural fixed point algorithm. In this algorithm, summaries are used to embody the effects of function calls. The summary of a function is called its *behavior* because it summarizes the visible behavior of the function. This means that the behavior consists of the data-flow facts in the final instruction of a function. The algorithm consists of the following four steps:

Step 1 The behavior of each function and each interrupt handler is determined using a data-flow analysis. In this step, an intraprocedural analysis is performed and function calls are ignored. This means that no data-flow facts are added through function calls and each function is analyzed only once.

Step 2 In the following step, the data-flow facts at each call-site are combined with the behavior of the callee. That is, a data-flow analysis is executed, where for each `call` instruction in a function f, the available data-flow facts are joined with the behavior of the called function. In this step, the behavior of f is extended and callers of f are reanalyzed in case the behavior of f has changed.

Step 3 An interprocedural fixed point iteration is conducted, starting from the main function of the program. Data-flow facts are propagated from each `call` instruction into the called function. Additionally, data-flow facts are propagated from all program location where interrupts are enabled into each interrupt handler. Callees and interrupt handlers are then reanalyzed with new data-flow facts available at the beginning of the function.

Step 4 Superfluous data-flow facts are removed at call-sites.

In each of the steps, all functions and interrupt handlers are analyzed using intraprocedural fixed point iterations. The steps mainly differ in the ways `call` instructions are handled and functions need to be reanalyzed.

In **Step 1**, each function and interrupt handler is analyzed exactly once in order to determine its behavior. The behavior is computed using an intraprocedural data-flow analysis as described before, but function calls are ignored. Then, in **Step 2**, each function and each interrupt handler is evaluated using an intraprocedural analysis. In this step, each function f is analyzed at least once, but the data-flow facts at each `call` instruction in f are joined with the behavior of the called function. Consequently, the results in the `call` instruction in f become larger. If more data-flow information is available in the final instruction of f, which corresponds to the behavior, all functions calling f have to be reanalyzed. Data-flow facts from interrupt handlers are propagated into all program locations where the corresponding interrupt is enabled. Hence, an interprocedural fixed point iteration is performed.

In **Step 3**, the analysis starts with the main function of the program. Analysis results are propagated from each `call` instruction of the main function into a called function f. If the data-flow facts present at the entry of f have

changed, then f needs to be reanalyzed based on the new data-flow facts using an intraprocedural fixed point iteration. As before, data-flow facts are propagated into all functions called from f. All in all, this leads to an interprocedural fixed point iteration. In like manner, data-flow facts are propagated from an instruction into each interrupt handler if interrupts are enabled in this instruction. This execution order has the advantage that no definitions are propagated from a calling function into another calling function, and hence, context-sensitivity is preserved. The same applies for the propagation of data-flow facts through interrupt handlers.

So far, no analysis results at call-sites have been overwritten using the behavior, even though the callee could, for instance, overwrite a register on every execution path. This is due to the join-operation at `call` instructions in **Step 2**, which joins the data-flow facts at the entry of a `call` instruction with the data-flow facts coming from the called function. From the behavior of a function after **Step 1** or **Step 2**, it is not visible whether there exists a path through the function on which a data-flow fact is generated or whether it is generated on all paths through that function. **Step 4** is executed in order to tackle this source of imprecision. This step is basically a repetition of **Step 2**, but a modified join-operation is executed at call-sites. If a data-flow fact was propagated into a called function in **Step 3** but is not available at the exit node of the respective function, then this observation implies that the data-flow fact was overwritten on all paths through the called function. Consequently, superfluous data-flow facts are removed at the corresponding `call` instruction and are replaced with the behavior of the function, leading to increased precision.

In **Step 1**, **Step 2**, and **Step 3**, only additional information is collected. That is, after each iteration of each step, the analysis results become larger or remain unchanged. This is in contrast to **Step 4**, where the analysis results become smaller than at the end of **Step 3**, while still an over-approximation is preserved. **Step 4** is only required if the data-flow fact also contains information where it was generated, which is required only in RDA. For GIFA and LVA, only the first three steps are executed.

5.2. Representation of Hardware Dependencies

As explained in Sect. 3, I/O registers control the behavior of the microcontroller. Reading or writing an I/O register often influences the behavior of the microcontroller. For instance, if bit 0 – called `TOIE0` – of the timer interrupt mask register `TIMSK` is set to one and interrupts are enabled, the timer/counter 0 overflow interrupt is enabled. Moreover, reserved bits should be written to zero if accessed in order to ensure compatibility with future devices. Another example can be seen in dependencies between different I/O registers when using I/O ports. Each port has three associated registers: a data register `PORTx`, a data direction register `DDRx`, and a port input register `PINx`, where `x` is the name of the I/O port. If the n-th bit of `DDRx` is one, then the n-th bit of `PINx` is configured as an output pin. If it is zero, the n-th bit of `PINx` is configured as an input pin, and holds a nondeterministic value when it is read. Changing `PORTx` can, for instance, activitate the pull-up resistors connected to the corresponding pin, depending on the configurations of `PINx` and `DDRx`.

In order to account for such details of the microcontroller, hardware dependencies are represented using a so-called *dependency map* in [MC]SQUARE. The dependency map contains for each I/O register a list of effects caused by accessing the corresponding register. The entries in the dependency map are used during static analysis in order to precisely model the influence of instructions on the behavior of the microcontroller. This information is stored once and reused whenever an instruction accesses an I/O register, which provides a generic model and simplifies the implementation of the analyses. All dependencies are described in the ATMEL ATmega16 datasheet [2].

5.3. Analyses

In the following, the different static analyses implemented in [MC]SQUARE are detailed. These analyses use the intra- and interprocedural approaches detailed before. Moreover, they use the dependency map in order to account for behavior caused by accessing certain I/O registers.

5.3.1. Stack Analysis

The only analysis independent of the results of all other analyses is STA. In binary code, the stack is frequently used to temporarily store the contents of working registers used in a function. In the beginning, the contents of these registers is pushed onto the stack, and at the end of the function, the contents of these registers is taken back from the stack and written into the corresponding registers. Hence, for a data-flow analysis it looks as if this function depends on the values of these registers, although the function does not use the values. Moreover, the stack is used

to store return addresses from function calls or interrupt handlers. STA is an intraprocedural analysis used to check two conditions. First, it is used to check which values are actually used in a function. Second, it checks whether the values stored on the stack are written into their corresponding source registers. This implies that the value of the register is the same before and after the corresponding `push` and `pop` instructions. This analysis corresponds to the *well-behavedness analysis* of runtime stacks described by Linn et al. [39].

Due to the dynamic nature of stacks, the size and contents of the stack at a specific program location can only be determined during runtime. In Fig. 5, an example is depicted that demonstrates the stack usage for storing working registers. Here, register `r1` is only used as a temporary variable, and at the end of the function it contains the same value as in the beginning of the function. A standard data-flow analysis that does not model the stack cannot recognize this. To solve this problem, and hence obtain more accurate results for other data-flow analyses, an abstract interpretation [12] is used to determine for each program location the set of possible stack configurations. Here, an abstraction of all possible stack configurations is propagated through each function. In each function, a check is performed whether the local stack configuration has not changed when the function is exited. The abstract interpretation observes all accesses to the stack such as `push`, `pop`, changing the stack pointer, and write accesses into the memory area of the stack. Moreover, it determines if at the end of the function the original values of the working registers are restored. The outcome of STA is a set of triples, which consist of the address of the `push` instruction, the address of the `pop` instruction, and the register. If STA fails due to an infinite number of possible stack configurations, for example, caused by loops or manual changes of the stack pointer, it is assumed that this function changes the contents of the complete SRAM and all working registers used within the function.

In the example shown in Fig. 5, STA correctly recognizes that at location `0x26` the original value of `r1` is restored and the value of `r1` is not modified. That is, the outcome of the analysis of this code fragment is {(`0x23`, `0x26`, `r1`)}. If the instruction at address `0x26` were replaced by `pop r0`, the stack analysis would fail because the value stored on the stack would not be written back into the original source register. That is, the register configuration would have changed.

```
0x23: push r1        ; store r1 on the stack
0x24: in r1 PORTA    ; read value from input port
0x25: out PORTB r1   ; write value to output port
0x26: pop r1         ; restore value of r1
0x27: ret            ; return from function
```

Figure 5: Store intermediate values on the stack

5.4. Reaching Definitions Analysis

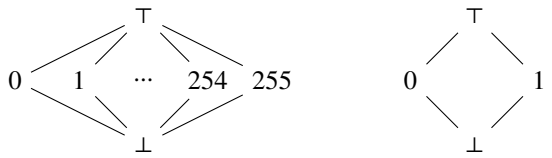
RDA computes for each program location and each memory location the set of program locations that may have written the value of the given memory location. Formally speaking, given a CFG $G = (V, E)$, the analysis produces for each $p \in V$ a set of pairs consisting of program locations $p' \in V$ and memory locations $m \in \mathbb{N}$. RDA can be defined as an equation system using a transfer function ω^{RDA} as follows:

$$
\begin{aligned}
RDA_{entry}(p) &= \begin{cases} \bot & : p \text{ is initial} \\ \bigsqcup\{RDA_{exit}(p') | (p', p) \in E\} & : \text{otherwise} \end{cases} \\
RDA_{exit}(p) &= \omega^{RDA}(p, RDA_{entry}(p)) \\
\omega^{RDA}(p, l) &= (l \setminus kill^{RDA}(p)) \cup gen^{RDA}(p)
\end{aligned}
$$

Here, $kill^{RDA}(p)$ denotes the set of reaching definitions overwritten in instruction p, while $gen^{RDA}(p)$ denotes the reaching definitions generated in p. Intuitively speaking, if an incoming definition is overwritten in p, then it it is removed from the data-flow facts and a new definition is generated. The analysis is conducted as an interprocedural fixed point iteration based on the algorithm described in Sect. 5.1.3.

Moreover, our approach applies an abstract interpretation to directly perform an extended form of constant propagation. Reaching definitions are annotated with the respective value. In case instructions are observed that write a

fixed value into a register, the reaching definitions are annotated with this value. For instance, if an instruction `eor r0 r0` (exclusive-or) is executed, then `r0` always contains the value `0` afterwards. In a similar way, if an instruction `add r0 r1` is executed and the reaching definitions of `r0` and `r1` are annotated with exact values, then RDA also infers the precise value of `r0` after this instruction. In case the results are ambiguous, however, it is assumed that any possible value can be written in order to generate an over-approximation.



(a) Register-wise (b) Bit-wise

Figure 6: Value representation using lattices

In this analysis, all registers except the SREG are represented using the lattice depicted in Fig. 6(a). The SREG is modeled bit-wise using the lattice depicted in Fig. 6(b). Many instructions of the ATMEL ATmega16 alter only single bits of the SREG. Instruction such as `cli` and `sei`, for instance, only set or clear the global interrupt flag, but no other bit of the SREG is touched. Arithmetic instructions such as `add` set certain bits of the SREG, for example, the negative flag or the zero flag, depending on the outcome of the operation, but the global interrupt flag remains unchanged. While it is often not possible to infer the exact value of all bits, it can be done for certain bits of the SREG.

In Fig. 7, the transfer functions used in the abstract interpretation are given for some instructions of the ATMEL ATmega16. Here, the value of a register `r` in the corresponding program location is denoted by $\|\mathtt{r}\|$. The instruction `ldi r c` loads a constant value `c` into register `r`. The instruction `mov r s` copies the value of register `s` into register `r`. For an `add r s` instruction, which sums up the values of `r` and `s` and stores the result in `r`, the precise value of the destination register can be computed if both the values of `r` and `s` are known at this location. Similarly, instructions such as `eor` (exclusive-or) are handled. In case `eor r r` is executed, the value can be directly inferred, even if the exact value of register `r` itself is unknown.

$$
\begin{aligned}
\texttt{ldi r c} &= \texttt{c} \\
\texttt{mov r s} &= \|\mathtt{s}\| \\
\texttt{add r s} &= \begin{cases} \|\mathtt{r}\| + \|\mathtt{s}\| & : \|\mathtt{r}\| \neq \bot \neq \|\mathtt{s}\| \wedge \|\mathtt{r}\| \neq \top \neq \|\mathtt{s}\| \\ \top & : \|\mathtt{r}\| = \top \vee \|\mathtt{s}\| = \top \\ \bot & : \|\mathtt{r}\| = \bot \vee \|\mathtt{s}\| = \bot \end{cases} \\
\texttt{eor r s} &= \begin{cases} 0 & : r = s \\ \top & : \|r\| = \top \vee \|s\| = \top \\ \bot & : \|r\| = \bot \vee \|s\| = \bot \\ r \otimes s & : \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 7: Instruction-specific transfer functions used in RDA

RDA uses the results of STA in order to obtain more precise analysis results than possible without knowledge about stack usage. Consider the example depicted in Fig. 5. Here, a function stores the value of `r1` on the stack by executing `push r1` and restores its value using `pop r1` before the function is exited. In this case, the reaching definition generated through the `pop r1` instruction can be removed and safely be replaced by the reaching definition for `r1` in the `push r1` instruction because the value of `r1` was not altered in-between. Consequently, STA leads to more concise reasoning about the origins of values because it allows for filtering of definitions that stem from storing

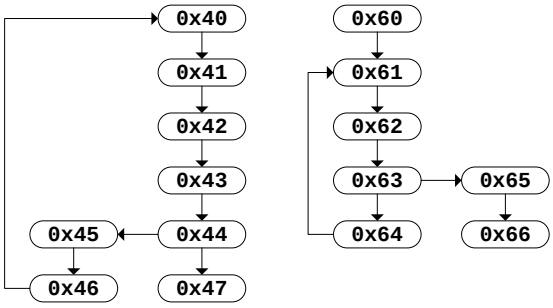
intermediate values, which are not changed.

Furthermore, RDA also depends on GIFA. If interrupts are enabled at a given program location, all reaching definitions steming from interrupt handlers have to be added to this program location. This means that increased precision in GIFA leads to smaller results of RDA. This exemplifies the dependencies between different static analyses.

In the following, an example for RDA using the program given in Fig. 8 is described. Here, two functions are used, which are located at program locations `0x40` and `0x60`. The function `0x60` is called by the function located at address `0x40`. In both functions, nondeterministic values are read from the environment through the input pins `PINA`, `PINB`, and `PINC`. The results of RDA are depicted in Tab. 1. For clarity, only the reaching definitions for `r0` are presented and the memory location `r0` is omitted. That is, an entry `0x42` in the table represents a reaching definition (`0x42`, `r0`). Furthermore, as the values of `r0` depend on nondeterministic input, the value analysis always infers $\top$, and hence, the value annotations are omitted as well. Each column presents the results after executing each step of the interprocedural fixed point iteration (cp. 5.1.3). In each row, the incoming data-flow facts for the respective instruction are depicted.

```
0x40: in r1 PINA
0x41: in r0 PINB
0x42: add r0 r1
0x43: cpi r0 32
0x44: brne 2
0x45: call 0x60
0x46: jmp 0x40
0x47: ret
...
0x60: in r1 PINC
0x61: add r0 r1
0x62: cpi r0 64
0x63: brne 1
0x64: jmp 0x61
0x65: inc r1
0x66: ret
```

(a) Program using two functions



(b) Corresponding CFGs

Figure 8: Example program to depict interprocedural RDA

The behaviors of the functions generated in **Step 1** are {`0x42`} for function `0x40` and {`0x61`} for function `0x60`, respectively. In **Step 2**, the results at the `call` instruction at program location `0x45` are combined with the local behavior of function `0x60`, which leads to definitions `0x42` and `0x61` at the exit of the call-instruction. In the following **Step 3**, data-flow facts from call-sites are propagated into called functions. Since only a single function call is present in this example, only the results of function `0x60` are extended. In **Step 4**, the analysis results are reduced. A check is performed whether the definition `0x42`, which is present at the entry of instruction `0x45`, is also included in the return statement of the called function. As this is not the case, the definition is removed, which leads to smaller results. Consequently, register `r0` has exactly one reaching definition at each program location.

5.5. Global Interrupt Flag Analysis

The global interrupt flag, which is stored in the highest bit of the SREG, defines whether interrupts are globally enabled or disabled. Without an analysis that determines the value of the global interrupt flag, it is assumed that interrupts are enabled at every program location in order to generate an over-approximation. If an interrupt handler reads a certain register and interrupts are active at any program location, this register can never be reset by DVR.

The status of the global interrupt flag is represented using the lattice depicted in Fig. 6(b). In order to obtain precise results, an abstract interpretation is applied that determines the status of the global interrupt flag for each program location. This abstract interpretation observes all accesses to the SREG done via instructions `cli` and `sei` and direct/indirect write accesses.

Table 1: Results of RDA for register r0 and the program given in Fig. 8

Instruction	Step 1	Step 2	Step 3	Step 4
0x40	0x42	0x42,0x61	0x42,0x61	0x61
0x41	0x42	0x42,0x61	0x42,0x61	0x61
0x42	0x41	0x41	0x41	0x41
0x43	0x42	0x42	0x42	0x42
0x44	0x42	0x42	0x42	0x42
0x45	0x42	0x42	0x42	0x42
0x46	0x42	0x42,0x61	0x42,0x61	0x61
0x47	0x42	0x42,0x61	0x42,0x61	0x61
0x60	∅	∅	0x42	0x42
0x61	∅	∅	0x42	0x42
0x62	0x61	0x61	0x61	0x61
0x63	0x61	0x61	0x61	0x61
0x64	0x61	0x61	0x61	0x61
0x65	0x61	0x61	0x61	0x61
0x66	0x61	0x61	0x61	0x61

With respect to dependencies between different static analyses, first of all, GIFA depends on STA. Consider the example given in Fig. 9. This code fragment depicts parts of an interrupt handler. When an interrupt handler is entered, the global interrupt flag is automatically cleared by the hardware. This means that the global interrupt flag is cleared in instruction 0x24, where the SREG is stored on the stack. When the value of the SREG is read from the stack in instruction 0x3c and written back into the SREG in instruction 0x3d, the global interrupt flag is still cleared. Without the information that the value pushed onto the stack and the value read from the stack are equal, the static analysis would have to assume that interrupts are possibly enabled.

```
0x23: in r0 SREG    ; write SREG into r0
0x24: push r0       ; store value on the stack
...
0x3c: pop r0        ; read value of SREG from stack
0x3d: out SREG r0   ; write back into SREG
...
0x44: reti          ; return from interrupt handler
```

Figure 9: Restore SREG in an interrupt handler

Moreover, sequences of instructions such as the program fragment depicted in Fig. 10 are frequently found in compiler-generated code in order to reset the SREG. Here, first r0 is set to 0 before the value is written into the SREG. Consequently, the global interrupt flag is cleared in program location 0x41. The value written into the SREG in instruction 0x41 is extracted from value annotations computed using RDA.

```
0x40: eor r0 r0     ; r0 = 0x00
0x41: out SREG r0   ; sreg = 0x00, i.e., interrupt flag cleared
```

Figure 10: Initialization sequence for the SREG in compiler-generated code

5.6. Live Variable Analysis

LVA is an analysis that determines for each program location the set of variables that may be read on some execution path through the program before they are overwritten [28]. These variables are called *alive* because their

value may be required in some execution. Its results are influenced by the results of GIFA and its precision strongly influences the effectivity of DVR.

In contrast to the analyses described before, LVA is a backward data-flow analysis. That is, the analysis traverses the CFG of the program in reverse order. Once an instruction is visited that reads a certain variable, this variable is added to the set of live variables. In like manner, a data-flow fact for a variable is removed in a certain program location if it is overwritten in the corresponding instruction. Given a CFG $G = (V, E)$ and $p \in V$, LVA can be expressed using the following equation system:

$$\begin{aligned} LVA_{exit}(p) &= \begin{cases} \bot & : p \text{ is final} \\ \bigsqcup\{LVA_{entry}(p') | (p, p') \in E\} & : \text{otherwise} \end{cases} \\ LVA_{entry}(p) &= \omega(p, LVA_{exit}(p)) \\ \omega^{LVA}(p, l) &= l \setminus kill^{LVA}(p) \cup gen^{LVA}(p) \end{aligned}$$

Note the different ordering compared to the equation system described in Sect. 5.1.2 in order to propagate data-flow facts back-to-front. Here, $gen^{LVA}(p)$ generates a data-flow fact for a variable m iff m is read in p. Similarly, $kill^{LVA}(p)$ contains the set of variables that are written in p.

As described before, functions are all program fragments reachable via `call` statements. Additionally, interrupts are also handled like functions. In binary code, functions do not have formal parameter values. Communication with functions is done via global variables, globally accessible registers, the runtime stack, or indirect loads and stores from and to a memory area indicated by a pointer register. The latter case is seldom used and leads to an over-approximation in this approach as indirect loads and stores can access all memory locations. The most common case is the usage of global variables, registers, and the stack. To handle functions and interrupt handlers, a local behavior is defined for them (cp. Sect. 5.1). The behavior of a function regarding LVA is a set containing all memory locations alive at the beginning of the respective function. In the worst case, this approach leads to an over-approximation of the behavior of a function by assuming that all memory locations are alive due to indirect reads, but in most cases few memory locations are alive. This analysis benefits from all analyses described before because without these analyses, the results obtained during LVA would be too inaccurate to be of use for DVR. The different steps of our interprocedural approach are conducted as detailed before.

Similar to the way RDA depends on STA, LVA depends on STA. If a value of a register is only stored on the stack and then read from the stack, the value is never really used. Consequently, pairs of `push` and `pop` instructions that have been identified using STA are not considered for LVA. Moreover, LVA also depends on GIFA. A memory location is alive in a program location if interrupts are enabled and the memory location is read in an interrupt handler. Consequently, limiting the set of program locations where interrupts are enabled leads to smaller LVA results.

As an example, consider the code fragment given in Fig. 8. LVA leads to the results shown in Tab. 2, which details the sets of live variables at the entry of each instruction. After **Step 1**, register `r0` is included in the local behavior of function `0x60` since it is alive at the entry of the function. In the first step, `r0` is not alive in the instructions located at `0x44` and `0x45`. In **Step 2**, the local behavior of function `0x60` is combined in the call instruction located at address `0x45`. Consequently, `r0` is now alive in instructions `0x44` and `0x45`. **Step 3** does not lead to different results and execution of **Step 4** is not required for LVA.

Step 3 in our interprocedural approach is required in order to obtain an over-approximation. In case the instruction `in r0 PINB` were replaced with an instruction that also reads `r0`, such as `sub r0 r1`, the register `r0` would also be alive in `0x40` and `0x46`. In this case, the alive register `r0` would have to be propagated into the exit of function `0x60` in order to prevent the function `0x60` from resetting `r0`.

6. Abstraction Techniques

This section details two abstraction techniques, namely DVR and PR, for the ATMEL ATmega16 microcontroller. Each of these abstraction techniques leads to state-space reductions in model checking. Moreover, they can be combined. While DVR can always be applied, PR only preserves CTL*-X, that is, validity of the X (next) operator is lost.

Table 2: Results of LVA for the program given in Fig. 8

Instruction	Step 1	Step 2	Step 3
0x40	∅	∅	∅
0x41	{r1}	{r1}	{r1}
0x42	{r0, r1}	{r0, r1}	{r0, r1}
0x43	{r0}	{r0}	{r0}
0x44	∅	{r0}	{r0}
0x45	∅	{r0}	{r0}
0x46	∅	∅	∅
0x47	∅	∅	∅

Instruction	Step 1	Step 2	Step 3
0x60	{r0}	{r0}	{r0}
0x61	{r0, r1}	{r0, r1}	{r0, r1}
0x62	{r0, r1}	{r0, r1}	{r0, r1}
0x63	{r0, r1}	{r0, r1}	{r0, r1}
0x64	{r0, r1}	{r0, r1}	{r0, r1}
0x65	{r1}	{r1}	{r1}
0x66	∅	∅	∅

6.1. Dead Variable Reduction

DVR copes with the state-explosion problem by reducing the number of states generated during state-space construction. If two states differ only in the value of a *dead* variable, that is, a variable whose value is never read again, then both states can be seen as equivalent, and hence, can be merged into a single state. In order to preserve the validity of the model checking results, variables used within the specification are never reset.

6.1.1. Algorithm

Yorav and Grumberg [45] define a variable to be *fully dead* at a given program location p if on every execution path starting from p, the variable is overwritten before it is read again. In contrast to their approach, we do not consider *partially dead* variables, which are variables that are dead on some execution paths. Following from the definition, DVR can be seen as the dual of LVA. That is, a variable that is not alive is dead, and hence, can be reset. Executing LVA is needed in order to compute the set of live variables. During model checking, however, variables need to be reset only once at program locations where they *die*. Dying locations are those program locations where the variable was alive in a preceeding location and then becomes dead. Resetting dying variables instead of dead variables reduces runtime requirements during model checking as there is no need to reset a variable twice.

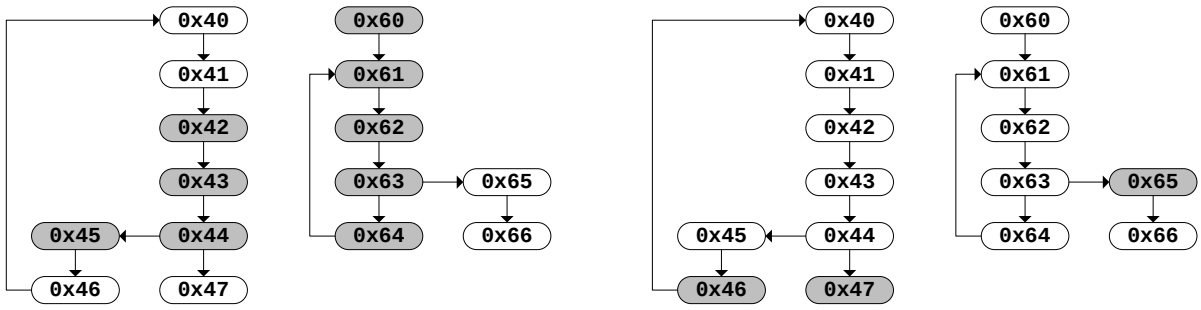
After the sets of live variables are determined, for each program location p the set D_p of dying variables has to be identified. This is done by successively comparing the sets of live variables of two consecutive program locations p and p'. The variables that are alive at p and are no longer alive at p' die at p'. Let $LVA(p)$ and $LVA(p')$ denote the sets of live variables at the exit of program locations p and p', respectively. Then, we have $D_{p'} = \bigcup\{LVA(p) | (p, p') \in E\} \setminus LVA(p')$. Furthermore, variables that are assigned a value in p' and not alive in any succeeding instruction are added to $D_{p'}$, thus considering the case that a value is never read. Afterwards, the variables accessed in atomic propositions used in the specification have to be removed from the set of dying variables. Finally, every program location p is annotated with its corresponding set D_p. This set indicates the variables that have to be reset during state-space construction.

6.1.2. Example

Consider the example program given in Fig. 8. Program locations where register `r0` is alive are depicted in Fig. 11(a). Similarly, the dying locations of `r0` are depicted in Fig. 11(b). The program location where `r0` dies are `0x46`, `0x47`, and `0x65`. In these instructions, `r0` was alive in a preceding location and then becomes dead. Hence, `r0` is reset whenever one of these instructions is visited during model checking.

6.2. Path Reduction

PR was first described by Yorav and Grumberg [45]. It is used to collapse single successor chains, which are computational paths consisting of states having only single successors, into a single step. This means that only the first and the last state of this path are stored in order to reduce states spaces and memory consumption. Furthermore, states are stored at program locations where the validity of the specification may be influenced. The disadvantage of this method is that it only preserves a divergence-sensitive stuttering bisimulation [3, 42] between the concrete and the abstract transition system. Consequently, it preserves CTL*-X, that is, validity of the next operator in CTL is lost. A formal proof is given by Yorav and Grumberg [45]. This restriction, however, is negligible due to rare use of the



(a) LVA (b) DVR

Figure 11: CFG from Fig. 8, results for LVA and DVR for register `r0` are emphasized

X operator in specifications when model checking binary code. As specifications are often based on the C code of the program and a single C statement is typically compiled into a sequence of different instructions, the X operator is barely used in practice.

6.2.1. Algorithm

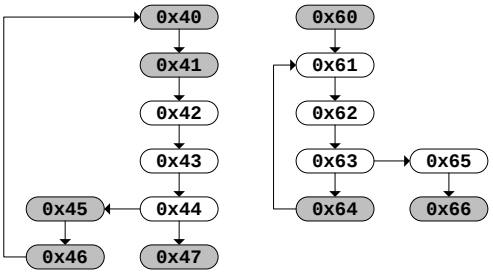
Our algorithm for PR consists of a static and a dynamic part in order to determine program locations of interest. These program locations are called *breaking points* by Yorav and Grumberg [45]. During state-space building, only states that are generated in program locations marked as breaking points are stored. First, program locations which satisfy certain conditions are determined by means of a static analysis. Yorav and Grumberg handle a parallel *while* language, and hence, all breaking points can be identified statically. In [MC]SQUARE, however, some of these breaking points have to be determined dynamically during state-space building due to indirect data accesses and nondeterminism. Yorav and Grumberg define the following program locations p to be breaking:

(i) p is the initial or terminating program location
(ii) p is associated with the program location of an assignment that changes a variable used within the formula
(iii) p is associated with the program location of a nondeterministic assignment
(iv) p is the head of a `while` statement
(v) p is labeled by a procedure call, or is the statement immediately following a procedure call
(vi) p is labeled by a communication statement (`send` or `receive`), or is the statement following inter-process communication

Condition (i) can be checked statically in [MC]SQUARE. Condition (ii) could be detected statically in binary code as well. Using a static approach, all instructions that indirectly write a memory location have to be marked as breaking, in addition to instructions that directly write memory locations used in the CTL formula. This includes frequently found constructs such as accesses to the stack using `push` instructions. Thus, detecting condition (ii) entirely statically leads to a coarse over-approximation and many instructions are unnecessarily marked as breaking. Consequently, direct writes are statically marked as breaking and indirect writes are resolved dynamically during state-space building in [MC]SQUARE.

For similar reasons, condition (iii) cannot be checked statically in [MC]SQUARE because nondeterminism is not indicated by certain statements in binary code. Different memory locations can introduce nondeterminism and are accessed through various instructions. Furthermore, a memory location can change back and forth between nondeterministic and deterministic behavior. For instance, an input port is switched to output or a timer is disabled. Such situations cannot be detected statically. Hence, a static analysis would lead to a too coarse over-approximation. Therefore, the third condition is checked dynamically during state-space building. This check is implemented by observing whether for the respective state more than one successor is generated.

Condition (iv) can be detected statically in binary code, but it requires special treatment because no explicit instructions for implementing loops exist. Loops are implemented using combinations of branching instructions and



(a) Breaking points

Instruction	Conditions	Instruction	Conditions
0x40	(i),(iii)	0x60	(iii)
0x41	(iii)	0x61	–
0x42	–	0x62	–
0x43	–	0x63	–
0x44	–	0x64	(iv)
0x45	(v)	0x65	–
0x46	(v)	0x66	(v)
0x47	(i),(v)		

(b) Satisfied conditions

Figure 12: Example for PR

unconditional jumps. Detecting loops is required in order to guarantee termination during state-space building. If there is a nonterminating loop in the program under verification without any breaking points on this loop, it would not be possible to detect revisits, and hence, the state-space building would not terminate. Termination of loops, however, cannot be detected. In order to ensure termination of state-space building, at least one program location on each loop is required to be a breaking point. Hence, all unconditional jumps targeting addresses lower than the address of the respective instruction and all indirect jumps are marked as breaking as well. Moreover, all branching instructions with negative offset are marked as breaking as these can lead to a loop. An instruction such as `brne -16`, for example, branches backwards in the program in case the zero flag in the SREG is set.

For condition (v), all call instructions including indirect and relative calls are statically marked as breaking. The targets of the respective call instructions are not breaking. Moreover, all `ret` instructions, which correspond to return statements in high-level programming languages, are marked as breaking. Marking `ret` instructions is required because the return address stored on the stack may have changed on the execution path from the function entry to the `ret` instruction. Hence, the instruction immediately following a `call` instruction is not marked, but the `ret` instruction leading to the immediately following instruction, because the called function may return to an instruction different from the original `call` instruction. Such behavior is not possible in the parallel *while* language considered by Yorav and Grumberg.

Condition (vi) is not directly applicable to binary code as it does not contain parallel processes. Parallelism is implemented by means of interrupt handlers, which show a comparable behavior (cp. Sect. 3). The interleaving of the main program and interrupt handlers, however, is asymmetric. That is, interrupt handlers can interrupt the main program but not vice versa. Moreover, there are no explicit communication statements that control the communication between the main program and interrupt handlers. Communication is performed using global variables. Whenever an interrupt handler is active, it can communicate with the main program by writing memory locations that are read by the main program. To represent this behavior, each location where interrupts may occur has to be breaking. Similar to condition (v), which handles `call` instructions, all `reti` instructions – return from interrupt handler – are statically marked as breaking points. Instructions where interrupts are enabled could be statically marked because GIFA delivers an over-approximation of the status of the interrupt flag. The exact status of the global interrupt flag, however, is always known during model checking. Hence, locations where interrupts are enabled are marked at runtime.

On first sight, it appears that PR would not provide significant benefit. In practice, however, interrupts are only active within certain parts of the program and inactive in most interrupt handlers. Often, interrupt handlers are implemented as long single successor chains, which particularly benefit from PR. This is shown in a case study presented in Sect. 7.

6.2.2. Example

Consider the program given in Fig. 8. Only those program locations emphasized in Fig. 12(a) are marked as breaking points. The conditions that apply at each program location are detailed in Fig. 12(b). In this program, we assume `0x40` to be the initial program location. Moreover, the instruction `0x40` contains a nondeterministic assignment because it reads a value from an input port, and thus, 256 successors are created. The same applies to

instruction 0x41. Instruction 0x45 is marked due to the call instruction. Instruction 0x46 is marked due to condition (iv) in order to detect loops. Instruction 0x47 is the final location in the program, and hence, condition (i) applies. In the other function, instruction 0x60 reads a nondeterministic value from the environment and is marked. Moreover, instruction 0x64 is marked due to condition (iv). A ret instruction is located at 0x66, and hence, this instruction is marked due to condition (v). No interrupt handlers are used, and thus, condition (vi) has no effect. Condition (ii) is not applied as no specification is used in this example.

7. Case Study

This section describes a case study using seven different programs for the ATMEL ATmega16 microcontroller in order to show the effect of DVR and PR on state spaces. The case study was conducted on a SUN Fire x4600 M2 server equipped with eight dual-core AMD Opteron 8220 processors and 256 GB main memory. Although [MC]SQUARE supports parallel state space building algorithms as described by Brauer et al. [8], only one of the processors was used in this case study in order to generate unbiased results. The seven programs chosen for this case study were all written by students in lab courses, during diploma theses, or in exercises. These programs were written in C and compiled using AVR-GCC. None of the programs was intentionally written for being model checked.

The results of the case study are shown in Tab. 3. The first line shows the results for every program without applying any abstraction techniques. The second and the third line show the results using DVR and PR, respectively. The last line demonstrates the results when both abstraction techniques are applied. The column *states stored* reflects the number of states that are stored in the state space. The column *states created* presents the number of states that are created during model checking. This number is typically higher than the number of states stored due to revisits. The column *size [MB]* shows the size of the state space. The last column shows the runtime needed for applying static analysis and building the complete state space. The invariant AG true was used in order to build the complete state space. Only small programs, for which state spaces could be built without applying any abstraction techniques, were used in order to compare the effects of the described techniques.

The first program called light_switch is a simple program utilized to demonstrate basic microcontroller functions. It consists of 162 lines of code. It uses two timers and no interrupts. In this program, DVR lowered the number of states stored by 22.26%. PR lowered the number of states stored by 78.65%, but it increased the number of states created by 164.01%. This is because of long single successor chains of which only the last state was stored. In order to detect revisits, the complete chain had to be visited again. For this small example, the runtime was increased because all static analyses were executed for DVR, which was not the case for the default configuration. Using DVR and PR together led to 84.65% less states stored compared to the application of no abstraction techniques. The reductions caused by both abstraction techniques did not add up completely, but their combination had a noticeable effect. The reduction in the number of states stored did not directly carry over to reduction in terms of memory requirements on this small example. The hash table used to store the state space is always initialized with a default size larger than the state space of the complete program. Consequently, 0.5 MB is the minimal memory requirement of [MC]SQUARE during model checking.

The next program called plant controls a fictive chemical plant. It consists of 225 lines of code, and it uses one timer and two interrupts. DVR had no effect in this program because the same variables are used throughout the complete program including the interrupt handlers. Therefore, this program has no location where a variable becomes dead, and hence, no variables were ever reset. PR lowered the number of states stored by 90.29% and increased the number of states created by 13.96%. This means that either the number of revisits was smaller compared to the program light_switch or the length of single successor chains was shorter. Applying PR led to a significant decrease in memory usage, that is, it dropped from 33.6 MB to 3.0 MB.

The program called reentrance is used to demonstrate a reentrance problem. The program itself is rather small. It consists of only 148 lines of code and uses one interrupt handler. A 16-bit variable i is accessed both in the main program and in the interrupt handler. Values are assigned using different instructions that access only 8 bits of i, but access to i is not protected. This lack of protection of a shared variable leads to invalid values of i with respect to the specification. Similarly to the program plant, DVR had no influence for the same reasons. Again, PR significantly reduced the state space and stinted 90.77% of the states stored. The number of states created was only increased by 10.85% due to few revisits. The memory requirements dropped from 23.5 MB to 2.2 MB.

The program `traffic_light` was written by students during a lab course. As the name suggests, it is used to control the operation of a traffic light. It comprises 155 lines of code. Moreover, two interrupts and one timer are used. Once again, DVR had no effect. PR showed a performance similar to the programs described before with an increase in the number of states created of 19.68%. The number of states stored were reduced by 89.62% and the memory requirements were reduced from 2.3 MB to 0.5 MB. Again, this is the lower bound in terms of memory requirements when using [MC]SQUARE.

A controller for a powered window lift used in a car was implemented in the program `window_lift`. This program was inspired by a real automotive task. The program we chose for this case study contains 289 lines of code, and it uses three interrupts and two timers. DVR alone had a significant effect on the state space as the number of states stored dropped from 2,342,564 to 307,176, which is a reduction of 86.89%. The required runtime dropped from 30.11 seconds to just 9.32 seconds. PR produced comparable results as the number of states stored dropped to 318,626. The number of states created, however, was increased by 47.43%. This explains the increased runtime when only PR was used. In combination, both analyses led to a state space consisting of only 40,048 states, which is a reduction of 98.29% compared to the original state space. When DVR and PR were applied, 494,140 states had to be created, which is a reduction of 80.91% with respect to the original values. Runtime requirements dropped from 30.11 seconds to 10.73 seconds. The memory requirements were reduced significantly; only 9.5 MB of memory were needed compared to 573 MB using the default configuration.

Table 3: Effects of DVR and PR one seven microcontroller programs

Program	**Options used**	**States stored**	**States created**	**Size [MB]**	**Time [s]**
`light_switch` 162 lines	none	4,268	6,296	1.2	0.11
	DVR	3,318	5,119	1.0	0.33
	PR	911	16,622	0.5	0.55
	both	655	13,521	0.5	0.52
`plant` 225 lines	none	130,524	135,949	33.6	2.11
	DVR	130,524	135,949	33.6	5.71
	PR	12,679	154,921	3.0	1.62
	both	12,679	154,921	3.0	4.85
`reentrance` 148 lines	none	107,649	110,961	23.5	1.22
	DVR	107,649	110,961	23.5	1.91
	PR	9,935	123,003	2.2	1.22
	both	9,935	123,003	2.2	1.91
`traffic_light` 155 lines	none	9,998	10,506	2.3	0.14
	DVR	9,998	10,506	2.3	1.71
	PR	1,038	12,563	0.5	0.49
	both	1,038	12,563	0.5	1.70
`window_lift` 289 lines	none	2,342,564	2,589,665	573	30.11
	DVR	307,176	335,724	73	9.32
	PR	318,626	3,818,060	77	40.43
	both	40,048	494,140	9.5	10.73
`can` 383 lines	none	147,259,483	154,271,836	34,552	2,317
	DVR	147,259,483	154,271,836	33,994	2,410
	PR	21,037,058	180,748,382	4,539	2,239
	both	21,037,058	180,748,382	4,724	2,209
`vector` 930 lines	none	47,477,797	48,419,003	11,522	812
	DVR	43,306,447	44,247,653	10,632	784
	PR	3,131,994	55,584,435	754	670
	both	3,131,031	55,583,472	754	720

A four-channel speed measurement, where communication with peripherals is performed using a CAN bus interface, was implemented in the program `can`. The program consists of 383 lines of code and uses two interrupt handlers. Using the default configuration, 147,259,483 states were stored. DVR did not cause any reduction because interrupts are active in every program location and indirect reads are used in the interrupt handlers. Using PR, the number of states stored dropped to 21,037,058, for which 4,724 MB of main memory are required. This is a reduction of 85.71%. The number of states created increased by 17.16%, but the runtime was slightly decreased.

The program `vector` reads various inputs from the environment and performs some geometric computations on these inputs. It consists of 930 lines of code but does not use interrupt handlers. Using DVR led to a minor reduction of the number of states stored as indirect reads are often used in this program. This prevented DVR from resetting more memory locations. Using PR, the number of states stored decreased from 47,477,797 to 3,131,994, which is a reduction of 93.40%. Using PR resulted in no major reductions in terms of runtime, but memory requirements dropped from more than 11 GB to 754 MB. Consequently, this program could also be checked using a conventional desktop computer, which is not possible without PR. The number of states created increased by 14.80%. Applying both DVR and PR had almost no additional effect.

Summarizing, it can be seen that PR reduces the number of states stored strongly for every program, and hence, memory consumption. On the other hand, PR does not lower the runtime because of the larger number of states created due to revisits. For some programs, the runtime was even significantly increased. As each state stored using [MC]SQUARE requires up to 2 kB of memory, space is our main concern. Consequently, this analysis should be used whenever possible.

The efficiency of DVR strongly depends on the structure of the analyzed program. Whenever there is a tight coupling between data variables across functions and interrupt handlers, DVR does not reveal any effect. Moreover, presence of indirect reads strongly reduces the number of program locations where memory locations can be reset in order to ensure an over-approximation. This is a known problem of static DVR techniques. While the runtime overhead caused through the static analysis increases the runtime for small programs such as `light_switch` or `plant`, this overhead is more than offset by the reduced runtimes due to smaller state spaces, as can be observed for the program `window_lift`. For many programs, state spaces are reduced using DVR.

Overall, these results show that abstraction techniques based on static analysis can significantly reduce state spaces during model checking. Furthermore, they reduce memory requirements. The runtime overhead caused through static analysis is moderate and pays off when model checking large programs.

8. Conclusion & Future Work

This article describes two abstraction techniques employed to tackle the state-explosion problem, and the underlying static analysis framework. Both abstraction techniques were previously used in other model checkers such as SPIN, ESTES, and MURPHI, but could not be transferred one-to-one to [MC]SQUARE due to the peculiarities of binary code. Interprocedural analyses that cope with these peculiarities are employed in [MC]SQUARE. While DVR is performed entirely statically, the preparation of programs for PR combines static and dynamic techniques due to interrupts and nondeterminism.

The results of DVR are comparable to the results achieved in other model checking tools, although the analysis has to be performed interprocedurally. On the other hand, PR has a higher impact on state spaces compared to other model checkers due to the structure of binary code. Similar results were observed by Quiros [30]. Binary code tends to have long single successor chains, which need not to be stored completely. For instance, a single C instruction could be compiled into a sequence of six instructions. Another source of single successor chains is interrupts. In most cases an interrupt handler cannot be interrupted by interrupts, and hence, it can be reduced efficiently. Both reduction techniques can be used to lower the size of state spaces. DVR can be used in any case as it does not have any effect on the validity of specifications. PR can only be used if the X operator in CTL is not needed.

A negative effect of using PR is an increased number of created states due to revisits. From our point of view, it is preferable to trade time for space because memory requirements are a bigger problem. In summary, it can be said that reduction techniques using static analysis can be used to tackle the state-explosion problem in explicit state model-checking. Significant improvements can be observed when using DVR and PR for model checking binary code. The impact PR has in this specific domain is even higher than in model checkers working on intermediate languages.

Judging from the size of a program alone, estimating sizes of state spaces is infeasible due to several effects. State spaces strongly depend on the number of input values and their domains. Concurrency introduced by interrupt handlers aggravates the state-explosion problem as well. The effectiveness of DVR and PR is difficult to estimate by inspecting the binary code manually because the program locations where states are stored and variables are reset strongly influence state spaces. In the past, we have successfully verified programs consisting of up to 5,000 instructions using [MC]SQUARE. On the other hand, we had to stop the verification process for some programs consisting of only 100 instructions after approximately 6,000,000,000 (partly symbolic) states had been created. From our point of view, a further combination of static and dynamic state-space reduction methods appears to be a feasible approach in order to pave the way for the industrial application of binary code model checkers such as [MC]SQUARE.

Apart from the development and implementation of the described algorithms for the ATMEL ATmega16, we have also implemented these techniques for other platforms such as the Intel MCS-51 [33]. Our experiences show that, in order to make such techniques applicable to binary code, details of the specific target platform have to be taken into account and special analyses have to be developed in order to handle certain architectural features. For instance, the Intel MCS-51 supports register bank switching in contrast to the ATmega16. Four register banks can be selected using the register selection bits. In order to compute precise results for RDA, LVA, and DVR, a register bank analysis has to be performed in order to compute a safe approximation of the register selection bits. Without register bank analysis, no reaching definitions could ever be removed because it would not be clear which register bank is active, and hence, which register is overwritten. In order to compute an over-approximation, the analysis would have to assume that all register banks could be written. In like manner, LVA and DVR are influenced by register banks.

Different hardware platforms, however, pose different challenges to static analysis and model checking. In particular, any verification argument for the ATmega16 must pay special attention to the fact that GPRs and IORs can be overwritten using indirect stores, which is not possible on the MCS-51. All in all, the MCS-51 requires more analyses, which are simple in their structure, while the ATmega16 requires fewer analyses using more sophisticated techniques.

One of the remaining challenges in static analysis of microcontroller binary code concerns indirect reads, writes, jumps, and calls. In order to compute the target locations of such constructs, a pointer analysis is required. Knowing the values of pointers renders DVR more precise. For example, an indirect read can access all memory locations, which prevents these locations from being reset. We believe that the performance of DVR can be significantly improved by embodying a pointer analysis. In contrast to high-level languages, inferring pointer values in binary code strongly depends on the ability to precisely analyze loops. In compiler-generated code on the ATMEL ATmega16, for example, all global variables are initialized in a loop during startup, where the corresponding memory locations are accessed using indirect reads and writes. Compared to high-level code, less semantic information about types and loop conditions is available in binary code. Few approaches have been developed that are suitable for computing loop conditions and loop invariants for low-level and binary code [19–21, 27].

References

[1] Atmel Corporation, July 2008. 8-bit AVR Instruction Set.
URL `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`

[2] Atmel Corporation, June 2008. Datasheet: ATmega16.
URL `http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf`

[3] Baier, C., Katoen, J.-P., 2008. Principles of Model Checking. The MIT Press.

[4] Balakrishnan, G., Reps, T., Melski, D., Teitelbaum, T., 2008. WYSINWYX: What you see is not what you execute. In: Verified Software: Theories, Tools, Experiments (VSTTE 2005), Zurich, Switzerland. Vol. 4171 of Lecture Notes in Computer Science. Springer, pp. 202–213.

[5] Ball, T., Rajamani, S. K., 2000. Bebop: A symbolic model checker for boolean programs. In: SPIN Model Checking and Software Verification (SPIN 2000), Stanford, USA. Vol. 1885 of Lecture Notes in Computer Science. Springer, pp. 113–130.

[6] Beckers, J., Klünder, D., Kowalewski, S., Schlich, B., 2008. Direct support for model checking of abstract state machines by utilizing simulation. In: Abstract State Machines, B and Z (ABZ 2008), London, UK. Vol. 5238 of Lecture Notes in Computer Science. Springer, pp. 112–124.

[7] Bozga, M., Fernandez, J.-C., Ghirvu, L., 1999. State space reduction based on live variables analysis. In: Static Analysis (SAS 1999), Venice, Italy. Vol. 1694 of Lecture Notes in Computer Science. Springer, pp. 164–178.

[8] Brauer, J., Schlich, B., Kowalewski, S., 2009. Parallel and distributed invariant checking of microcontroller software. Electronic Notes in Theoretical Computer Science 254, 45–63, 4th International Workshop on Systems Software Verification (SSV 2009).

[9] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., Veith, H., 2001. Progress on the state explosion problem in model checking. In: Informatics - 10 Years Back. 10 Years Ahead. Vol. 2000 of Lecture Notes in Computer Science. Springer, pp. 176–194.

[10] Clarke, E. M., Grumberg, O., Peled, D. A., 1999. Model Checking. The MIT Press.

[11] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, Zheng, H., 2000. Bandera: Extracting finite-state models from java source code. In: International Conference on Software Engineering (ICSE 2000), Limerick, Ireland. ACM, pp. 439–448.
[12] Cousot, P., Cousot, R., 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages (POPL 1977), Los Angeles, USA. ACM, pp. 238–252.
[13] Davey, B. A., Priestley, H. A., April 2002. Introduction to Lattices and Order. Cambridge University Press.
[14] Emerson, E. A., 1991. Handbook of Theoretical Computer Science. Vol. B. The MIT Press, Ch. Temporal and Modal Logics, pp. 995–1072.
[15] Hecht, M. S., 1997. Flow Analysis of Computer Programs. Elsevier.
[16] Heljanko, K., May 1997. Model checking the branching time temporal logic CTL. Research Report A45, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland.
[17] Holzmann, G. J., 1999. The engineering of a model checker: The GNU i-protocol case study revisited. In: Theoretical and Practical Aspects of SPIN Model Checking. Vol. 1680 of Lecture Notes in Computer Science. Springer, pp. 232–244.
[18] Kildall, G. A., 1973. A unified approach to global program optimization. In: Principles of Programming Languages (POPL 1973), Boston, USA. ACM, pp. 194–206.
[19] King, A., Søndergaard, H., 2008. Inferring congruence equations using SAT. In: Computer Aided Verification (CAV 2008), Princeton, USA. Vol. 5123 of Lecture Notes in Computer Science. Springer, pp. 281–293.
[20] King, A., Søndergaard, H., 2010. Automatic abstraction for congruences. In: Verification, Model Checking, and Abstract Interpretation (VMCAI 2010), Madrid, Spain. Vol. 5944 of Lecture Notes in Computer Science. Springer, pp. 197–213.
[21] Kosakai, T., Maeda, T., Yonezawa, A., 2007. Compiling C programs into a strongly typed assembly language. In: Advances in Computer Science (ASIAN 2007), Doha, Qatar. Vol. 4846 of Lecture Notes in Computer Science. Springer, pp. 17–32.
[22] Leven, P., Mehler, T., Edelkamp, S., 2004. Directed error detection in C++ with the assembly-level model checker StEAM. In: Model Checking Software (SPIN 2004), Barcelona, Spain. Vol. 2989 of Lecture Notes in Computer Science. Springer, pp. 39–56.
[23] Lewis, M., Jones, M., 2006. A dead variable analysis for explicit model checking. In: Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2006), Charleston, South Carolina. ACM, pp. 48–57.
[24] Marlowe, T. J., Ryder, B. G., 1990. Properties of data flow frameworks - a unified model. Acta Informatica 28 (2), 121–163.
[25] Mehler, T., 2005. Challenges and applications of assembly-level software model checking. Ph.D. thesis, Universität Dortmund.
[26] Mercer, E., Jones, M., 2005. Model checking machine code with the GNU debugger. In: Model Checking Software (SPIN 2005), San Francisco, CA, USA. Vol. 3639 of Lecture Notes in Computer Science. Springer, pp. 251–265.
[27] Morrisett, J., Walker, D., Crary, K., Glew, N., May 1999. From System F to typed assembly language. ACM Trans. Program. Lang. Syst. 21 (3), 527–568.
[28] Nielson, F., Nielson, H. R., Hankin, C., 1999. Principles of Program Analysis. Springer.
[29] Noll, T., Schlich, B., 2008. Delayed nondeterminism in model checking embedded systems assembly code. In: Hardware and Software: Verification and Testing (HVC 2007), Haifa, Israel. Vol. 4899 of Lecture Notes in Computer Science. Springer, pp. 185–201.
[30] Quirós, G., March 2006. Static byte-code analysis for state space reduction. Master's thesis, RWTH Aachen University.
[31] Regehr, J., Cooprider, N., 2007. Interrupt verification via thread verification. Electronic Notes in Theoretical Computer Science 174 (9), 139–150.
[32] Regehr, J., Reid, A., Webb, K., 2003. Eliminating stack overflow by abstract interpretation. In: Embedded Software (EMSOFT 2003), Philadelphia, USA. Vol. 2855 of Lecture Notes in Computer Science. Springer, pp. 306–322.
[33] Reinbacher, T., Brauer, J., Horauer, M., Schlich, B., 2009. Refining assembly code static analysis for the Intel MCS-51 microcontroller. In: Industrial Embedded Systems (SIES 2009), Lausanne, Switzerland. IEEE Computer Society Press, pp. 161–170.
[34] Schlich, B., June 2008. Model checking of software for microcontrollers. Dissertation, RWTH Aachen University, Aachen, Germany.
URL `http://aib.informatik.rwth-aachen.de/2008/2008-14.pdf`
[35] Schlich, B., Brauer, J., Wernerus, J., Kowalewski, S., 2009. Direct model checking of PLC programs in IL. In: Dependable Control of Discrete Systems (DCDS'09), Bari, Italy. Accepted for publication.
[36] Schlich, B., Kowalewski, S., 2006. [mc]square: A model checker for microcontroller code. In: Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006), Paphos, Cyprus. IEEE Computer Society Press, pp. 466–473.
[37] Schlich, B., Löll, J., Kowalewski, S., 2008. Application of static analyses for state space reduction to microcontroller assembly code. In: Formal Methods for Industrial Critical Systems (FMICS 2007), Berlin, Germany. Vol. 4916 of Lecture Notes in Computer Science. Springer, pp. 21–37.
[38] Schlich, B., Rohrbach, M., Weber, M., Kowalewski, S., 2006. Model checking software for microcontrollers. Tech. Rep. AIB-2006-11, RWTH Aachen University, Aachen, Germany.
URL `http://aib.informatik.rwth-aachen.de/2006/2006-11.pdf`
[39] Schwarz, B., Debray, S., Andrews, G., Legendre, M., 2001. PLTO: A link-time optimizer for the Intel IA-32 architecture. In: Workshop on Binary Translation (WBT 2001), Barcelona, Spain.
[40] Self, J. P., Mercer, E. G., 2007. On-the-fly dynamic dead variable analysis. In: Model Checking Software (SPIN 2007), Berlin, Germany. Vol. 4595 of Lecture Notes in Computer Science. Springer, pp. 113–130.
[41] Sharir, M., Pnueli, A., 1981. Program Flow Analysis: Theory and Applications. Prentice Hall, Ch. 7 (Two Approaches to Interprocedural Data Flow Analysis), pp. 189–234.
[42] van Glabbeek, R., Weijland, W., 1996. Branching time and abstraction in bisimulation semantics. Journal of the ACM 43 (3), 555–600.
[43] Vergauwen, B., Lewi, J., 1993. A linear local model checking algorithm for CTL. In: CONCUR'93, Hildesheim, Germany. Vol. 715 of Lecture Notes in Computer Science. Springer, pp. 447–461.
[44] Vitek, J., Horspool, R. N., Uhl, J. S., 1992. Compile-time analysis of object-oriented programs. In: Compiler Construction (CC 1992), Paderborn, Germany. Vol. 641 of Lecture Notes in Computer Science. Springer, pp. 236–250.
[45] Yorav, K., Grumberg, O., 2004. Static analysis for state-space reductions preserving temporal logics. Formal Methods in System Design 25 (1), 67–96.